DStore: An in-memory document-oriented store

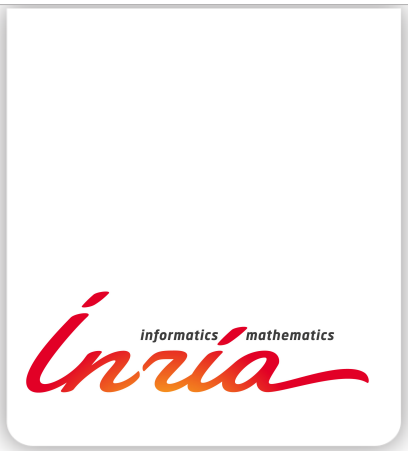
Viet-Trung Tran, Dushyanth Narayanan, Gabriel Antoniu, Luc Bougé

DStore: An in-memory document-oriented store

Viet-Trung Tran, Dushyanth Narayanan, Gabriel Antoniu, Luc Bougé

RESEARCH REPORT
N° 8188
December 2012
Project-Teams KerData

ISSN 0249-6399 ISRN INRIA/RR--8188--FR+ENG



DStore: An in-memory document-oriented store

Viet-Trung Tran*, Dushyanth Narayanan†, Gabriel Antoniu‡, Luc Bougé *

Project-Teams KerData

Research Report n° 8188 — December 2012 — 21 pages

Abstract: As a result of continuous innovation in hardware technology, computers are made more and more powerful than their prior models. Modern servers nowadays can possess large main memory capability that can size up to 1 Terabytes (TB) and more. As memory accesses are at least 100 times faster than disk, keeping data in main memory becomes an interesting design principle to increase the performance of data management systems. We design DStore, a document-oriented store residing in main memory to fully exploit high-speed memory accesses for high performance. DStore is able to scale up by increasing memory capability and the number of CPU-cores rather than scaling horizontally as in distributed data-management systems. This design decision favors DStore in supporting fast and atomic complex transactions, while maintaining high throughput for analytical processing (read-only accesses). This goal is (to our best knowledge) not easy to achieve with high performance in distributed environments. DStore is built with several design principles: single threaded execution model, parallel index generations, delta-indexing and bulk updating, versioning concurrency control and trading freshness for performance of analytical processing.

Key-words: In-memory, DStore, NoSQL, vertical scaling, document-oriented store, versioning

* ENS Cachan, Rennes, France
† Microsoft Research, Cambridge, United Kingdom
‡ INRIA, Rennes, France

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

DStore : un système de stockage orienté documents résidant en mémoire vide

Résumé : À la suite de l'innovation continue dans la technologie du matériel, les ordinateurs sont de plus en plus puissants que leurs modèles précédents. Les serveurs modernes de nos jours possèdent une grande capacité de mémoire vide dont la taille est jusqu'à 1 téra-octets (To) et plus. Puisque les accès en mémoire vide sont au moins 100 fois plus rapide que sur le disque dur, conserver les données dans la mémoire vide devient un principe de conception intéressant pour augmenter la performance des systèmes de gestion des données. Nous désignons DStore, un stockage orienté documents résidant en mémoire vide pour tirer parti de la haute vitess des accès en mémoire vide. DStore peut se mettre à l'échelle en augmentant la capacité de mémoire vide et le nombre de cœurs de CPU au lieu de faire mettre à l'échelle horizontalement comme dans les systèmes de gestion de données répartis. Cette décision de conception favorise DStore à soutenir des transactions complexes en assurant la rapidité et l'atomicité, tout en conservant un haut débit élevé pour le traitement analytique (seulement des lectures).

Mots-clés : DStore, NoSQL, Scalabilité verticale, stockage orienté documents, versionage, mémoire vide

1 Introduction

NoSQL movement

Over the past few years, NoSQL data stores have been widely adopted by major Internet companies, such as Amazon, Facebook, and Google. NoSQL rose in response to the observation that current relational database management systems (RDBMS) are not a good fit for storing and retrieving Internet-specific data. First, RDBMS have been built along the relational model that is considered to poorly meet the requirements of the current data models found in popular Internet applications. In some cases, the relational model is too powerful for data that have no associative relationship among them. For example when what is needed is simply to access a piece of data given a key (key-value model). In other cases, data from social network services rely on graph structures withs nodes, edges and properties to represent their complex relationships: the relational model becomes insufficient in this situation. Second, most RDBMS sacrifices scalability for ACID semantics, whereas popular Internet applications don't often need ACID for their data. For instance, Facebook status updates or tweets can be stale and there is no penalty to propagate those updates later under the guarantee that the propagation is done eventually.

NoSQL data stores trade the complexity of having ACID semantics and a relational data model for higher performance and higher scalability. Among different types of NoSQL data stores, *document-oriented stores* offer an interesting compromise between the system performance and the rich functionality on a document-oriented data model. While key-value stores operate on independent key-value pairs and RDBMS are optimized for structured data having associative relationships, document-oriented stores fall in the middle: data is packed in documents, which are similar to data rows in RDBMS but they are very flexible and self-organized. Each document consists of multiple data fields, but it does not need to follow a fixed schema that defines the structure of the document. Therefore, a document-oriented store can be considered as a semi-structured database.

Compared to key-value stores, document-oriented stores can be seen as the next logical step from storing simple key-value pairs to more complex and meaningful data structures that are encapsulated in documents. Similarly to key-value stores, each document can be accessed given a globally shared document ID. Moreover, document-oriented stores provide the capacity to query a document not only based on the document ID but also based on the document content itself.

Trending towards vertically scaling by leveraging large memory

As hardware technology is subject to continuous innovation, computer hardware has become more and more powerful than the prior model. Modern servers are often equipped with large main memory capabilities, which can have sizes up to 1 TB. Given this huge memory capability, data-management systems are now able to store their data in main memory, so that hard disks become unnecessary [1]. While memory accesses are at least 100 times faster than disk, keeping data in main memory becomes an obvious trend to increase the performance of data-management systems.

In a recent study conducted by Microsoft Research [2], memory is at an inflection point where 192 GB of memory cost only $2640. Further, the decreasing cost/size ratio of memory still benefits from the Moore's law, this resulting in the possibility to double the memory size with the same cost every 18 months. In this context, vertically scaling data management systems by using bigger memory is cost-effective.

Current disk-based data-management systems only leverage main memory as a caching mechanism. One way to scale those systems vertically (scale-up) is to increase the cache size so that more data can be cached in main memory for faster accesses. However, this optimization is

suboptimal, as it cannot fully exploit the potential of large memory. The scalability is limited as disk-based data-management systems have to implement complex mechanisms to keep data on cache and on disks consistent.

2 Goals

In the following section, we explore the potential benefits of the "scale-up" approach to the design of a scalable system. DStore aims to be able to scale up by adding more memory rather than scaling horizontally by adding more servers as in distributed data management systems. We design and implement DStore, a document-oriented store with the following goals.

Fast and atomic complex transaction support. Various document-oriented stores such as CouchDB [3] and MongoDB [4] offer only support for atomic access at the granularity of a single document. Complex transactions that access multiple documents are not guaranteed to be atomic. Because documents are distributed across different storage servers, support for atomicity of multiple document access may require distributed locks, which results in poor system performance.

Atomic complex transactions are required in various scenarios in many current applications. Consider a bank company where each user account is represented by a document in a document-oriented store. In order to complete a bank transfer, the following operations in multiple documents must be done in a single atomic query: (1) verifying if there are sufficient funds in the source account in the first document, (2) decreasing the balance of the source account, and (3) increasing the balance of the destination account in the second document. Without atomic guarantees, the database will fall in an inconsistent state. Similarly, multiple operations in a single document also need to be atomic such as conditional update, read before write, etc.

The main design goal of DStore is to support atomicity for complex transactions and to do so with low additional overheads.

High throughput for analytical processing. Analytical processing refers to read queries that may access multiple documents or the entire data store in order to get a summary report for analytic purposes. One typical example of analytical processing in DBMS is the scan query whose goal is to select any data records that satisfy a particular condition. In the situation where concurrent update and read queries may access the same pieces of data, synchronization for data consistency is unavoidable, which leads to a slow update rate and a low-throughput analytical processing.

To deliver high throughput for analytical processing without interfering with update queries, DStore must be able to isolate both types of workloads with low overhead.

Achieving these two goals, we thus claim that DStore can provide fast, atomic complex transaction processing that refers to the update rate in data writing, while it also delivers high throughput read accesses for analytical purposes. This claim will be validated through the design of DStore and some preliminary synthetic benchmarks at the end of this chapter. In the future, we plan to reenforce our claim by performing more benchmarks on real-life workloads.

3 System architecture

3.1 Design principles

In order to achieve the aforementioned goals, we design DStore with the following principles:

Single threaded execution model

To date, the search for efficient utilization of compute resource in multi-core machines triggered a new architecture for multi-threading applications. Previous application designs rely on multiple threads to perform tasks in parallel in order to fully utilize CPU resources. In reality, most of the tasks are not independent of each other, as either they access the same part of data, or some tasks need part of the results from another task. This well-known problem (concurrency control) made it nearly impossible to have full parallelization in multi-threading environments.

DStore targets complex transactions where each transaction consists of several update/read operations on multiple different documents. Let us consider the following example: Transaction T1 updates 3 documents A, B and C. Transaction T2 updates 3 documents B, C and D. Another transaction T3 has to read documents B and C and update document D. Since T1, T2 and T3 are not mutually independent, it is impossible to perform these transactions concurrently using multiple threads without synchronization with exclusive locks. Obviously, in the case of more complex transactions, there is a higher possibility that those transactions are not independent of each other.

As a result, DStore relies on a single-threaded execution model to implement only one thread (called the *master thread*) for executing all update transactions sequentially. This approach is first advocated in [5] as a result of trending in in-memory design. As long as one single thread is performing data I/O, thread-safe data structures are not necessary. In other words, the code for locking mechanisms in concurrency control can be completely removed without loosing correctness, which results in less execution overhead. A recent study in [1] showed that locking and latching mechanisms in the multi-threading model create nearly 30% overhead in data-management systems.

Parallel index generations

With its rich document-oriented data model, a document-oriented store can offer an interface that can be close to that of RDBMS. One of the nice features is the possibility to query a document not only based on the document ID but also based on the document contents itself. To enhance the query performance, both RDBMS and document-oriented stores such as CouchDB [3] and MongoDB [4] rely on indexes. Maintaining indexes allows fast lookup to desired documents, but usually creates a certain amount of overhead for updates and deletes. Particularly, if indexes are built by the same thread for update transactions (the *master thread*) in a synchronous fashion, the system performance will be reduced twice when doubling the number of indexes.

To speed up index generation and to reduce overhead on the *master thread* for update transactions, DStore assigns index generation task to dedicated background threads, called *slave threads*. Each *slave thread* manipulates one index by maintaining a data structure such as the *B+tree* data structure that we will present further in Section 4 . *B+tree* allows searches in $O(\log(n))$ logarithmic time. Because DStore resides in main memory, all indexes keep only pointers to the actual documents in order to minimize memory consumption. In this setting, an *in-place data modification* when updating a particular document is very expensive. Locking and synchronization among *slave threads* and with the *master thread* are needed to keep all indexes in consistent states.

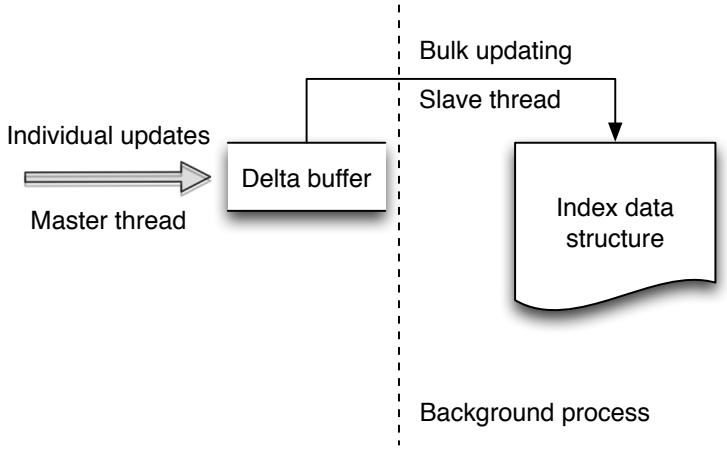


Figure 1: Delta indexing mechanism.

To avoid such a synchronization and to keep all the indexes independent from each other (even when they share the documents), *in-place data modifications* have to be avoided. DStore keeps each document immutable. Update transactions will not modify document contents, but rather create new documents and rely on indexes to commit the changes back to DStore. This mechanism is referred to as *copy-on-write* in the data-management community.

Delta-indexing and bulk updating

Since indexes are maintained by *slave threads*, the *master thread* that executes update transactions needs only to write new documents (in case of an update or an insert) to the memory space (part of the main memory reserved for holding DStore data), and to push an index request in the waiting queues of each index. This mechanism is referred to as *differential files* or *delta-indexing* [6] (Figure 1). DStore names the waiting queues as *the delta buffers*. Each queued element is typically a key-value pair, where the key is the indexed document attribute and the value is the pointer to the created document.

This *delta-indexing* mechanism allows DStore to potentially sustain high update rates under the expectation that pushing an index request to the *delta buffer* is faster than updating the data structure for the index itself. This expectation is obviously made possible as following: The *delta buffer* can be a simple queue that has the complexity of $O(1)$ for push operations whereas data structures for indexing such as *B+tree* need $O(\log(n))$ for every insert of lockup. Even when the *delta buffer* is a *B+tree*, the time to insert in the *delta buffer* is still lower due to its small size.

Moreover, one novel design choice we make is to leverage *bulk updating* to maintain indexes in background. This technique allows us to achieve three goals in DStore:

- First, bulk inputs in *delta buffers* are sorted before inserting into the indexes in order to better leverage cache behavior. As data is sorted, there is a high chance that inserting a new element in the *B+tree* will follow the same path from the *B+tree* root to the leaf or a partial path that was already cached in previous accesses.

- Second, merging the *delta buffer* to the index data structure in bulk avoids readers to read partial updates of a multi-key complex transaction. Thus, it guarantees transaction

atomicity. DStore implements a versioning mechanism to control the moment when the new versions of indexes are revealed to readers. In our design, this happens after all updates in the *delta buffer* are merged to the previous index.

- Third, the *delta buffer* can be compacted before being processed. As a sequence of inserts and deletes on the same documents may exist in the *delta buffer*, DStore can remove obsolete updates and keep only the latest update for each particular document. Therefore, DStore potentially minimizes the number of modifications to the persistent index data structure.

Versioning for concurrency control

Even if DStore has one *master thread* to execute update transactions sequentially, it allows multiple reader threads to run analytical processing concurrently. DStore relies on versioning for concurrency control to be able to deliver high-throughput for reading while minimizing interference with the *master thread* and with the *slave threads* building indexes.

DStore uses *B+tree* as the data structure for indexes and leverages shadowing (copy-on-write) [7] to guarantee that a set of updates are isolated in a new snapshot. Before a *slave thread* updates its index, it clones the index to create a newer snapshot and applies the updates only to that snapshot. When this process is completed, the new snapshot will be revealed to readers. Since the *slave threads* and readers access different snapshots of the indexes, they can work concurrently without locking.

Moreover, the novel idea in DStore is to merge updates to each index in bulk where only one new snapshot represents all the updates in the *delta buffer*. This approach has the advantage of reducing the number of intermediate snapshots in order to reduce memory consumption. This is clearly the difference between our approach and a pure copy-on-write implementation in *B+tree* [7], in which each update requires cloning an entire path from the *B+tree* root to the corresponding leaf to create a new snapshot.

Stale read for performance

Analytical processing can accept a certain level of staleness [8]. Results from a read query can be slightly out-of-date if they are used for analytic purposes. For example, social network websites such as Facebook allow users to write messages on their wall and get updates from other users. It is acceptable for such a query for all recent updates to return stale data which contain updates performed seconds to minutes ago.

To take advantage of the above property, DStore gives users the choice to decide the freshness level of an analytic query on a per-query basis. Instead of returning up-to-date data, a *stale read* only accesses the latest snapshot of the indexes. This choice leaves *stale reads* to be executed independently in an isolated fashion, at the cost of not being able to query data in the *delta buffers*. For a *fresh read*, locking is needed before scanning each *delta buffer* to avoid threading exceptions. Of course, this will impact negatively on the update rate of the *master thread* and also on the *slave threads*.

3.2 A document-oriented data model

Documents are the main data abstraction in DStore. Each document consists of multiple data fields but it does not need to follow a fixed schema that defines the structure of the document. Thus, it is very flexible and self-organized. For example, here is a document holding an employee information:

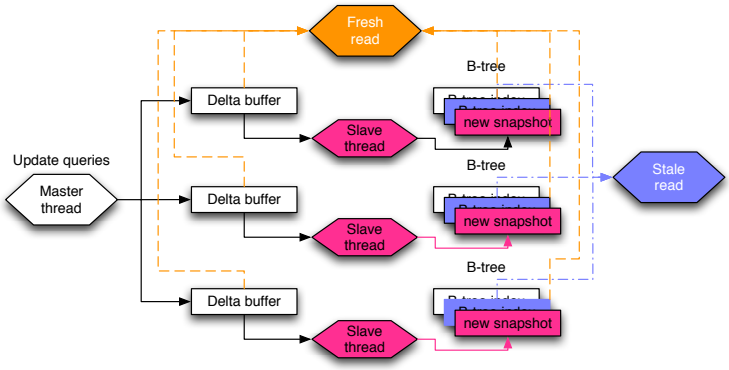


Figure 2: DStore service model.

ID: **int**
Name: **char**[30]
Address: **char**[60]

Because a document is flexible, it can contain more data fields for another employee as shown below:

ID: **int**
Name: **char**[30]
Address: **char**[60]
Passport: **int**

DStore allows users to access a document not only by the document ID, but also through several indexes. Assuming that *Name, Address* are globally unique, DStore can build three indexes for *ID, Name*, and *Address*.

3.3 Service model

As discussed in Section 3.1, DStore is built on the idea of having only one *master thread* to process update transactions sequentially but to allow concurrent analytical processing. Figure 2 represents the service model of DStore.

DStore executes *slave threads* in background for maintaining the *B+tree* indexes, where each *slave thread* is responsible for one index. Once the *slave thread* starts, it checks if the *delta buffer* is not empty, then it pushes all elements in the *delta buffer* to a new snapshot of its *B+tree* index. Only when finished, this snapshot is revealed for reading and the *slave thread* can then repeat the whole process for subsequent data updates.

Actually, each *delta buffer* consists of two parts: an *in-coming queue* for input coming from the *master thread* and a *readonly queue* holding updates that are under processing by the *slave thread* (Figure 3). When the *slave thread* starts building a new snapshot, the *in-coming queue* is flagged to be the *readonly queue* and a new *in-coming queue* is created. Basically, this mechanism is done to favor bulk processing and to minimize locking overheads on the *delta buffer*.

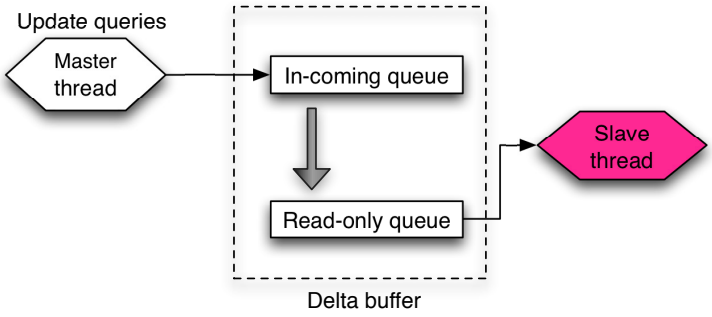


Figure 3: Zoom on the *delta buffer* implementation.

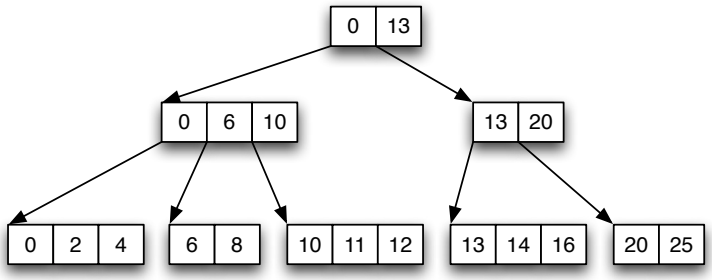


Figure 4: An example of a *B+tree*.

DStore allows users to configure which data structures will be used for the *delta buffers*. For instance, the *delta buffer* can be a *B+tree* that provides $O(\log(n))$ for both lookup and insertion. By default, DStore uses the vector datatype, as it allows faster insertion of $O(1)$ at the cost of slower lookup $O(n)$. This design decision favors the implementation of one *master thread* for updates in DStore.

4 DStore detailed design

4.1 Shadowing *B+tree*

DStore uses *B+tree* data structures for the indexes. The *B+tree* is a tree data structure for keeping data sorted for fast retrieval in logarithmic time. Each *B+tree* inner node contains entries that are mappings between keys and their children, while a *B+tree* leaf node contains key-value pairs. Keys in a *B+tree* follow a minimum-key rule: If a node N2 is a child of node N1, then any key in the child node N2 is bigger or equal to the key in N1 pointing to N2. Figure 4 represents a concrete example of a *B+tree* where each node has maximum 3 keys.

To favor concurrent accesses without using a locking mechanism, DStore *B+tree* implementation is inspired by the work presented in [7]. Unlike the original work, our *B+tree* is designed for one writer and many readers in order to eliminate locking overheads entirely in case of concurrent writers. DStore ensures that only one *slave thread* modifies the *B+tree*, so that there is no need to lock *B+tree* nodes during tree traversals and during the cloning process, as discussed in their

paper.

B+tree in DStore is configured to have between b and 2b + 1 entries per node and uses a proactive approach for rebalancing. During a tree traversal from root to leaves in an INSERT or a DELETE operation, a node with 2b + 1 entries is split before examining its child. When a node with b entries is encountered, either it has to be merged with its siblings or keys have to be moved from siblings into it. This proactive approach simplifies tree modifications as nodes will not be modified after visited. When shadowing, nodes are cloned in the order that the downward traversal is done from root to the leaves.

To enable shadowing in which different snapshots share *B+tree* nodes, each *B+tree* node has an internal reference counter that records how many parent nodes currently point to it. Figure 5 represents the example of the aforementioned *B+tree* with a reference counter in each node.

B+tree operations

Create. To create a new *B+tree*, a root node is initialized with zero entries and its reference counter is set to 1. The root node can contain less than b entries while all other nodes have to consist of b to 2b + 1 entries.

Clone. To clone a *B+tree* having the root node V1, the contents of V1 will be copied into a new root V2. This operation leads to the fact that any child of V1 is also referenced by the new root V2. Therefore, the reference counter in each child node of V1 has to be increased.

An example of this cloning process is presented in Figure 6. The reference counters of two nodes [0, 6, 10] and [13, 20] are set to 2.

Select. To lookup for a key in a *B+tree*, a downward tree traversal is needed. The algorithm starts examining the *B+tree* root of the desired snapshot and follows the appropriate inner node that covers the input key. When it reaches the leaf, the associate value of the selected key is returned if the key exists. Furthermore, during this downward traversal, no locking is required because tree nodes are immutable, except those of the snapshot under modification. DStore guarantees only read-only snapshots are visible for Select operations.

Insert. An Insert operation requires a lookup for the corresponding leaf while it has to clone all the nodes in the downward tree traversal. When a node N is encountered, the following procedure is executed:

- The reference counter is examined. If it is 1, meaning it only belongs to the current snapshot, there is no need to clone the node. Otherwise, the reference counter is greater than 1 and that node has to be cloned. This is done in order to avoid modifying nodes that also belong to other snapshots. The contents of this node are copied to a new node N' with the reference counter set to 1. The reference counter of the node N is decremented because N no longer belongs to the current snapshot. In addition, the reference counters in children of N are incremented to reflect the change that they have another new parent N'. The pointer in the parent node of N is now pointing to N'.
- N' is then examined under the proactive split policy. If it is full with 2b + 1 entries, it has to be split. If it has only b entries, it has to be merged with its siblings or some keys have to be moved from its siblings into it. As discussed, the proactive split policy prevents modifications from propagating up to parent nodes.

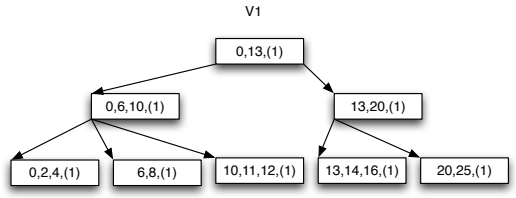


Figure 5: Initial *B+tree* V1.

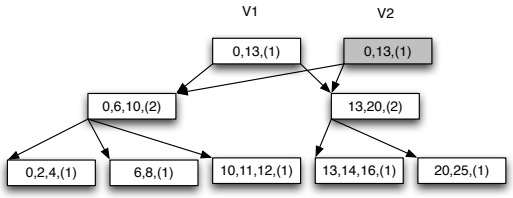


Figure 6: Creating a clone V2 of *B+tree* V1.

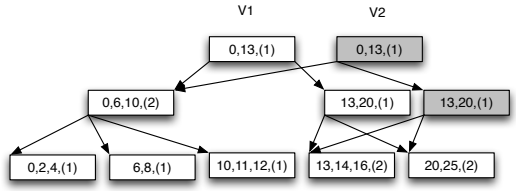


Figure 7: Node [13, 20] is cloned.

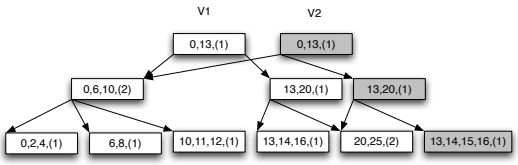


Figure 8: Leaf [13, 14, 16] is cloned. The new leaf is modified to host a new key 15.

For example, Figure 6 shows two snapshots V1 and V2 share tree nodes. When inserting a key 15 to the snapshot V2, the node [13, 20] is cloned first as shown in Figure 7. At the end, the leaf [13, 14, 16] is cloned and key 15 is added to the new leaf (Figure 8).

Delete. To delete a key, the same procedure as in Insert is executed. Nodes in the downward tree traversal are cloned and examined under a proactive merge policy. If a node that has the minimum number of b entries is encountered, the algorithm merges it with its sibling or moves entries into it. This policy guarantees that a node modification due to a Delete affects only its immediate parent.

4.2 Bulk merging

Using a *B+tree* implementation with a shadowing mechanism as presented in the previous section, it is easy to implement *bulk merging* in DStore. For each index, a *slave thread* creates a new clone of the latest snapshot of the *B+tree* index by using the *B+tree* Clone functionality. This operation is fast, as only a new root is created as a copy of the source *B+tree* root. Then, the *slave thread* starts examining all elements in the *delta buffer* to run the corresponding Insert or Delete operations on the new clone. When this process is done, the new clone is read-only and is revealed to readers. The *slave thread* then creates a new snapshot based on the new clone and the whole procedure loops again.

Our *bulk merging* approach reduces the number of intermediate snapshots and is potentially faster than a copy-on-write approach. In fact, our approach generates one single snapshot for multiple updates while a copy-on-write approach needs to clone the entire tree traversal from root to leaf for each Insert or Delete.

4.3 Query processing

We now discuss how DStore handles different kinds of queries.

Insert and Delete. Insert and Delete queries are executed sequentially by the *master thread*. Each Insert or Delete operation is translated into a series of messages that are fed into the *delta buffers*. Each message consists of a MESSAGE_TYPE that defines if it is an Insert or a Delete to the index, the value of the index key and a pointer to the new Inserted document or NULL in case of a Delete.

By default, DStore uses a vector datatype for *delta buffer*. When the *delta buffers* are not full, the cost for the *master thread* to finish one Insert or Delete is $m \times O(1)$ where m is the number of indexes. This is expected to be faster than $m * O(\log(n))$ in case the *master thread* has to manipulate the indexes himself (*B+tree* has $O(\log(n))$ complexity).

Read. DStore supports the following two types of Read queries.

Stale Read. DStore favors Stale Read operations in order to achieve high performance for analytic processing. A Stale Read accesses only the latest snapshots of the *B+tree* indexes that are generated by the *slave threads*. Thanks to the shadowing mechanism that ensures snapshot isolation, each Stale Read can be performed independently without any interference with the *slave threads*. Obviously, the *slave threads* are working on the newer snapshots of the indexes and will reveal them to readers only when finished.

Stale Reads achieve high performance at the cost of not accessing unindexed updates in the *delta buffers*. Thus, the staleness of results depends on the *delta buffers sizes* and the disparity between the update rate of the *master thread* and the processing rates of *slave threads*.

Obviously, Stale Reads on different indexes are independent and are executed in parallel.

Fresh Read. To guarantee the results of Fresh Read are up-to-date, both the *delta buffer* and the latest snapshot of the appropriate *B+tree* index needs to be accessed. Compared to Stale Read, a Fresh Read requires a lock on the *delta buffer* for scanning unindexed updates, thus it negatively impacts on the update rate of the *master thread*. However, we expect the cost for locking is minimal as the *delta buffer* is small size, the *readonly queue* is immutable, and only the *in-coming queue* needs to be locked.

Update. To perform this kind of query in DStore, a Fresh read is needed to select the desired document. This operation includes two steps: scanning the *delta buffer* and lookup the *B+tree* structure of the appropriate index. When it is done, the *master thread* can transform the original update query into a series of Delete and Insert pairs, one for each index. Its purpose is to delete the old index entry and insert a new one to reflect the update in the new snapshots.

Because only the *master thread* executes queries sequentially and any index is updated in bulk, update query is guaranteed to be atomic. For each index, a Delete and Insert pair is put atomically to the *delta buffer* so that any concurrent Fresh read will be aware of the atomic update. Additionally, since an Update is translated to a Delete, Insert pair, DStore avoids *in-place data modification* as Delete and Insert only affect an index. Therefore, DStore can achieve parallel index generation in which indexes are built independently by a number of *slave threads*.

One particular case is that some indexes may be updated faster than the others and thus Read query on those indexes may return more up-to-date results. However, this is not a problem because it does not break out the atomicity guarantee in DStore.

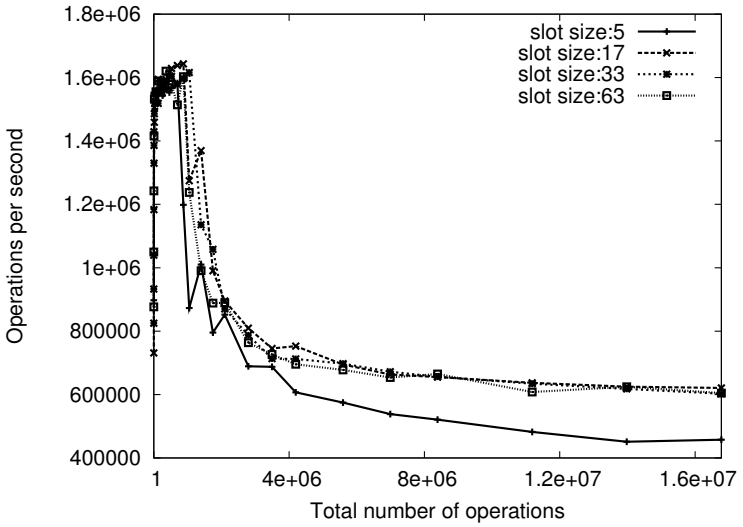


Figure 9: Impacts of *B+tree* slot size on DStore performance.

Complex queries. Complex query is a query involving more than one operation (Read, Insert and Delete, etc.) on more than one document. NoSQL stores often do not support this kind of queries in an atomic fashion [9]. DStore, on the other side, is capable of atomically handing any complex query, which is a combination of aforementioned Read, Insert and Delete operations. This guarantee is simply achieved through two mechanisms: (1) Insert multiple operations to *delta buffers* is atomic. (2) Every *B+tree* index structure is updated in bulk thanks to shadowing. Therefore, a Stale read will not see partial updates as long as it accesses a read-only snapshot. A Fresh read will not see partial updates either as long as the *master thread* can put updates of a complex query to each *delta buffer* atomically.

5 Evaluation

DStore is implemented from scratch in C++, using the Boost library [10] for threading and memory management.

We evaluate DStore through a series of synthetic benchmarks, focusing on its design principles presented in Section 3.1. Our experiments were carried out on the Grid'5000 testbed, on one node of the Parapluie cluster located in Rennes. The node is outfitted with AMD 1.7 Ghz (2 CPUs, 12 cores per CPU) and 48 GB of main memory.

Impact of the *B+tree* slot size on performance

In DStore, the *B+tree* slot size refers to the number of entries configured per *B+tree* node. Changing this value has an impact on the *B+tree* height, which defines the number of steps for a downward traversal from the root to a leaf (the *B+tree* height is equal to $\log_m n$ where m is the slot size and n is the total number of keys). If the slot size is too small, examining a tree node to find the pointer to the appropriate child is fast (binary lookup) but more nodes will be accessed before reaching the appropriate leaf. If the slot size is too big, tree traversal from root to leaf is fast as the tree height reduced but it will increase the time to access a *B+tree* node. Especially

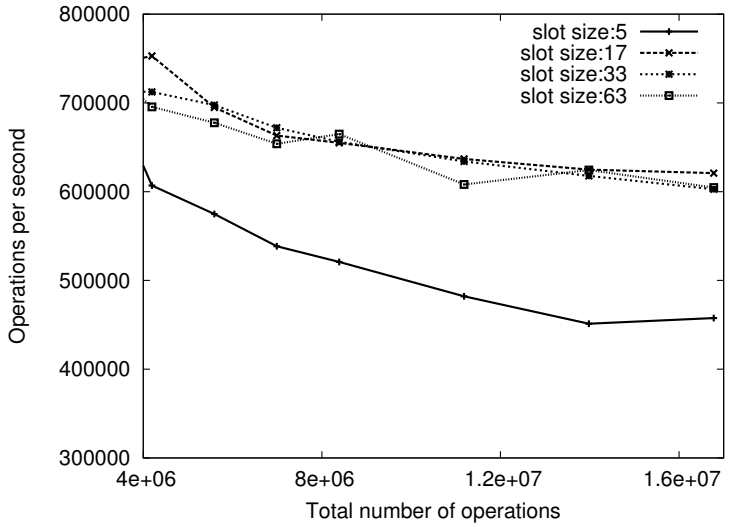


Figure 10: A Zoom in the impacts of *B+tree* slot size for large number of Insert operations.

for Insert or Delete operations, modifications on tree nodes that require merging or shifting keys are slower for bigger slot size.

Furthermore, the performance of the *B+tree* implementation with a shadowing mechanism in DStore is heavily influenced by the slot size. A bigger slot size means bigger *B+tree* nodes need to be cloned during shadowing. Thus, it increases the memory consumption as well as the time to copy contents from one node to another. As the slot size impacts the *B+tree* performance, it impacts directly the performance of DStore as well.

In the first experiment, we aim to evaluate the impact of the slot size on the performance of DStore. To this end, we configure DStore to build only one index. We start by inserting from 1 to 2^{24} distinct random integer key-value pairs to DStore and measure the completion time. We take the total number of Insert operations and divide it by the measured completion time to get the insert rate in terms of operations per second. The experiment was done for different slot sizes which are 5, 17, 33 and 63 entries. Each test has been executed 3 times and the average was taken into consideration.

The results are shown in Figure 9. As observed, DStore with slot size 17 achieves the best performance. Figure 10 is a zoom in for a clearer view of the impact. When slot size is bigger than 17, the cost for shadowing, merging *B+tree* nodes, shifting keys is getting to be higher than what is gained from reducing the tree traversal path. Thus, the performance of DStore did not increase when increasing the slot size.

Impact of sorted delta buffers

In this experiment, we evaluate an optimization we introduced in DStore when merging updates in *delta buffers* to the corresponding indexes. As discussed, the *delta buffer* is sorted in order to better leverage caching effects. With a sorted *delta buffer*, there is a higher chance that inserting or deleting an element in a *B+tree* will follow the same traversal path from the *B+tree* root to a leaf or a partial path that was already cached in previous accesses.

We conduct the same tests as in the previous experiment. We keep the slot size to be 17 and a maximum *delta buffer size* of 524288 elements. We measure the operations per second when

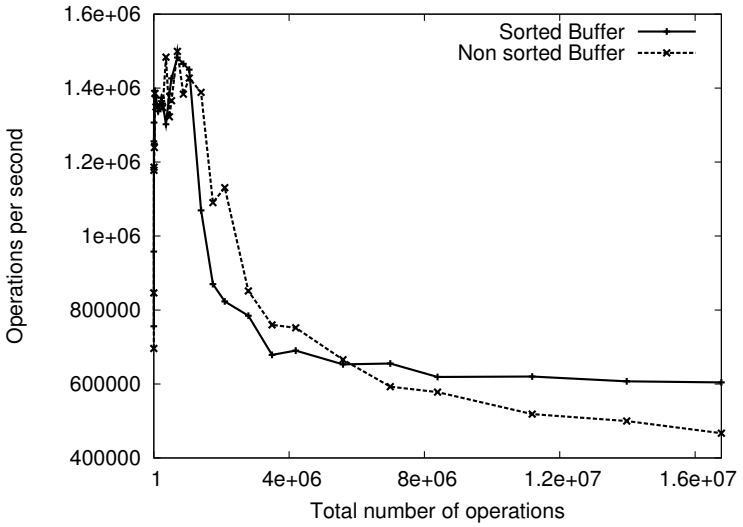


Figure 11: Performance of DStore in two cases: Sorted vs non-sorted delta buffers.

we insert 1 to 2^{24} randomly distinct integer key-value pairs to an empty DStore with only one index. Two cases are examined: *sorted delta buffer* vs *non-sorted delta buffer*.

As represented in Figure 11, DStore with *sorted delta buffer* performs less fast in the beginning, but outperforms the case of *non-sorted delta buffer* when *B+tree* size increases. This can be explained by the cost to sort the *delta buffer*. When the *B+tree* index is small, the caching effect is not significant, such that sorting the input ends up making DStore slower. When the *B+tree* is big, the benefits of caching effects are more significant than the initial effort to sort the *delta buffer*.

Comparing DStore to a pure *B+tree* implementation

We now aim to get a hint of how DStore performs compared to a pure *B+tree* implementation. Our third experiment measures the insert rate in terms of operations per second when we insert from 1 to 2^{24} randomly distinct integer key-value pairs to DStore and to a *B+tree* structure. Again, each test was run 3 times and the average of the results is taken into consideration (the standard deviation was low). Both the pure *B+tree* structure and *B+tree* implementation in DStore were configured to use a slot size of 17.

We can observe on Figure 12 that DStore performs slightly better. In the beginning, DStore outperformed the *B+tree* approach and this result is due to the *delta buffer*. Internally, each index of DStore is implemented in a producer-consumer model where the *master thread* keeps putting new operations in the *delta buffer* and the *slave thread* in turn takes all operations from the *delta buffer* to update its *B+tree* structure. Consequently, when the *delta buffer* is not full, the *master thread* can finish one insert operation in $O(1)$ time which is far better than $O(\log(n))$ in case of a pure *B+tree* implementation.

Further, as the *slave thread* cannot keep up with the *master thread*, the *delta buffer* gets full in a long-term run. This situation is shown in the right part of the Figure 12 where DStore performs only slightly better than the pure *B+tree* structure. Obviously, the *B+tree* implementation with shadowing mechanism in DStore should be slower than a pure *B+tree* due to the cost for shadowing. However, the obtained results can be explained by two reasons. First, DStore sorts

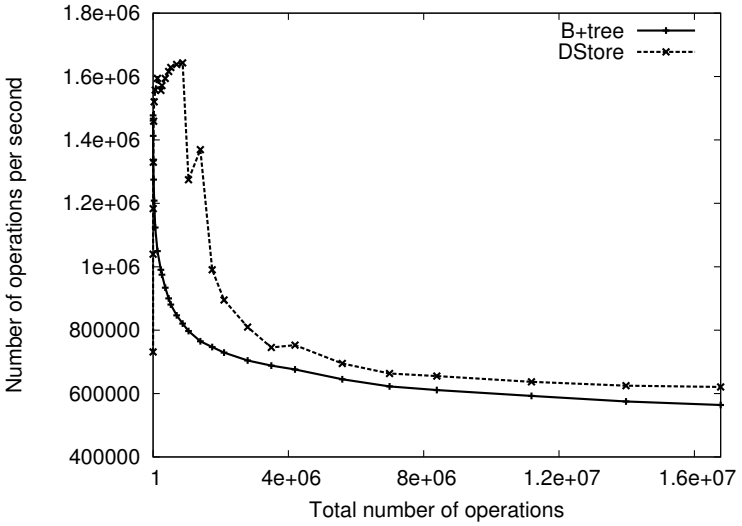


Figure 12: B+tree vs DStore: Slot size = 17.

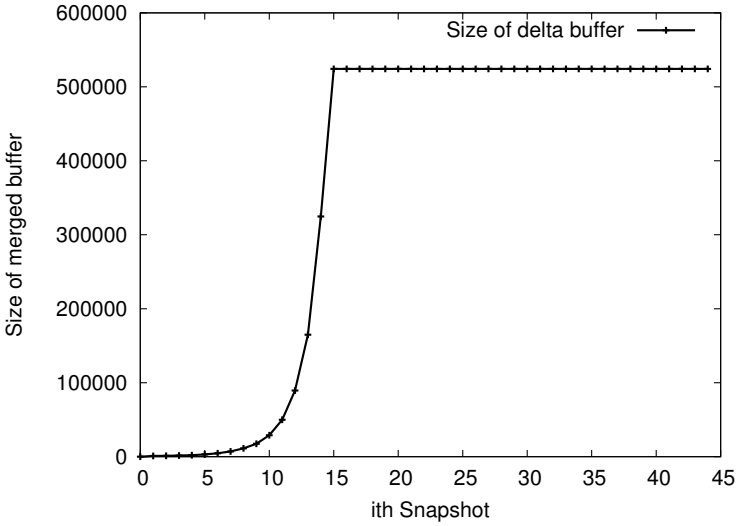


Figure 13: Evolution of the delta buffer.

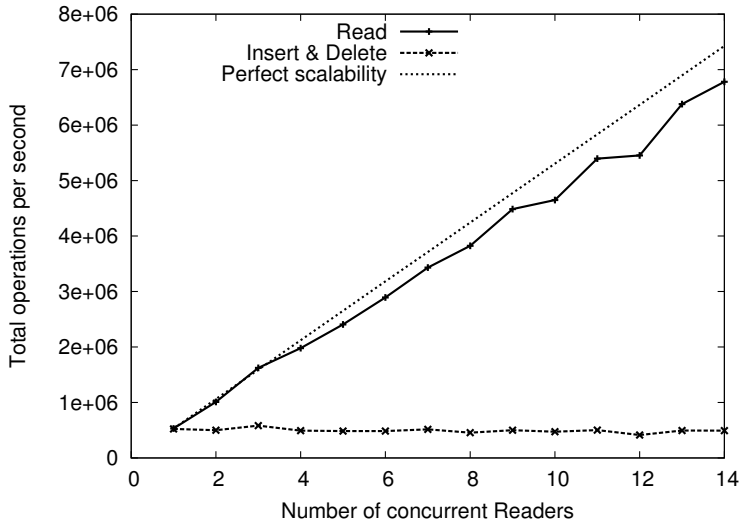


Figure 14: DStore performance in concurrency: Multiple readers and one master thread for inserts and deletes.

the *delta buffer* before merging to the *B+tree* so that it can leverage better the caching effects, as demonstrated in the previous experiment. Second, DStore performs the merging in bulk that creates only one new snapshot to reflect all updates in the *delta buffer*. This minimizes the number of *B+tree* nodes to be cloned and thus increases the shadowing performance.

Figure 13 gives more details with regard to the above arguments. When inserting 2^{24} randomly distinct key-value pairs to DStore, only 44 snapshots were created. The curve also shows the evolution of the *delta buffer size*. The *delta buffer* increased to the maximal configured value, when the speed of the *master thread* cannot be faster than that of the *slave thread*.

DStore performance under concurrency

One of the design goal of DStore is to provide high performance for both update queries in transactional processing and read queries in analytical processing. DStore supports one *master thread* for updates, but allows multiple read queries to be processed concurrently with the *master thread* as well. The idea is to leverage a shadowing mechanism to isolate read queries and update queries (Insert and Delete) in different snapshots, so that they can be processed independently in a lock-free fashion.

To evaluate DStore performance under concurrency, we design one experiment that starts by a warm-up phrase: 2^{24} distinct key-value pairs are inserting to DStore with one index. Then, we launch concurrent readers (up to 14), each of them performing Stale read operations that request 2^{24} keys from DStore. In the meantime, the *master thread* keeps inserting and deleting random keys with the purpose of constantly having 2^{24} records in DStore. We measure the number of operations per second for both the *master thread* and the readers.

As expected, Figure 14 demonstrates that DStore achieves a good scalability when increasing the number of concurrent readers. Moreover, there is very little overhead on the *master thread* as its performance does not decrease, but remains constant at about 600,000 operations per second.

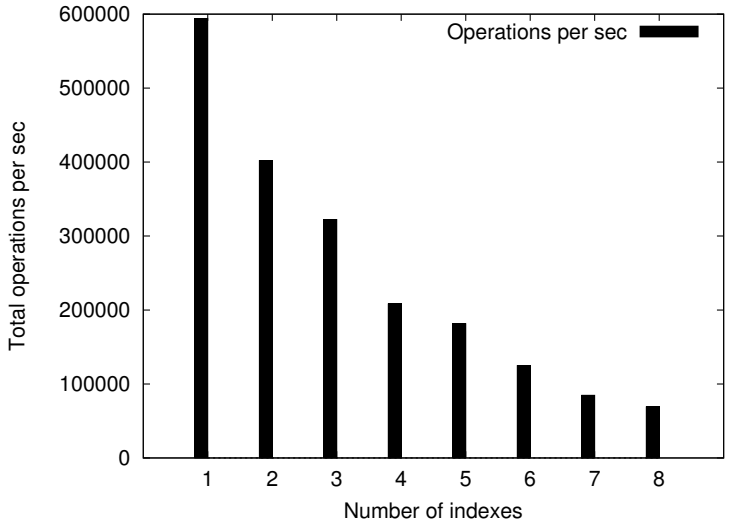


Figure 15: DStore performance when building multiple indexes.

Features	DStore	H-Store	HyPer	CouchDB
Document-oriented	Yes	—	—	Yes
In-memory	Yes	Yes	Yes	—
Versioning	Yes	—	—	—
Atomic complex query	Yes	Yes	Yes	—
Concurrent readers	Yes	—	Yes	Yes
Fresh Read	Yes	Yes	Unknown	No/Eventual Consistency
Stale Read	Yes	—	Yes	Yes
Bulk merging	Yes	—	Unknown	—

Table 1: A comparison between DStore, H-Store, HyPer and CouchDB.

Impact of building multiple indexes

DStore supports multiple indexes and it provides a mechanism to build those indexes in parallel. In this experiment, we measure the impact of building many indexes on the insert rate in terms of operations per second. For each test, we fix the number of indexes to be built in DStore and start inserting randomly-distinct 2^{24} key-value pairs to DStore. The completion time is measured and the insert rate in operations per second is calculated.

Figure 15 shows the insert rate decreases when increasing the number of indexes. The result was anticipated, as the more indexes have to be built, the more work the *master thread* has to do to finish one insert operation. In fact, it has to transform each operation into a series of corresponding operations for each *delta buffer*.

However, the performance of DStore in building multiple indexes is good as compared to the case where the indexes are built sequentially. In that setting, the performance must drop by a factor of 2 when doubling the number of indexes. DStore performance, on the other hand, decreased less than 50 % thanks to the delta indexing mechanism.

6 Related work

There are several related work as below.

H-Store. H-Store [11] is an experimental row-based relational database management system (DBMS) born in a collaboration between MIT, Brown University, Yale University, and HP Labs. H-Store supports fast transaction processing by leveraging main memory for storing data. To avoid the overhead of locking in multi-threading environments, H-Store follows single-threaded execution model where only one thread is used to execute transactions sequentially.

Compared to our approach, H-Store does not support high-throughput analytical processing. It is only optimized for online transaction processing (OLTP) and cannot execute read queries in parallel. Our approach serializes transaction processing on one single *master thread*, but allows multiple readers to access DStore concurrently without any interference with the *master thread*. Therefore, DStore has the potential to handle efficiently both fast update transactions and high-throughput analytic read queries.

Hyper. HyPer [12] is a main-memory database management system built with the purpose of being able to handle both online transaction processing (OLTP) and online analytical processing (OLAP) simultaneously. HyPer relies on a virtual memory snapshot mechanism that is assisted in hardware by the Operating System (OS) to maintain consistent snapshots for both OLAP and OLTP queries. Upon an OLAP request, HyPer clones the entire database and forks a new process using kernel APIs in the OS. This new process is then able to work on that consistent snapshot without any interference with the main database. In multi-core machines, multiple OLAP threads can be launched simultaneously as long as they only read a private snapshot.

HyPer shares many similarities with our system, but DStore differentiates from HyPer in many aspects. First, DStore is a document-oriented data store rather than a RDBMS. Its simplified document-oriented interface, which is close to the actual data models found in popular Internet applications, allows the system to achieve higher performance under that particular Internet workload [13]. Second, both DStore and HyPer leverage shadowing to separate transactional processing from analytical processing, but DStore does not clone the entire database. DStore implements a *B+tree* shadowing mechanism to clone only indexes and does so in a way that index cloning operations are done in parallel. Thus, our scheme minimizes memory consumption and is potentially faster than that of HyPer.

Moreover, DStore fully supports versioning as the direct result of its shadowing mechanism. DStore maintains all generated snapshots of each index and allows selecting any snapshot for reading purposes. Regarding HyPer, it does not provide a versioning functionality due to an expensive cost of cloning the entire database.

CouchDB. CouchDB [3] is a document-oriented store under the Apache License. Unlike our system, CouchDB does not leverage main memory. It was designed to *scale out* in distributed environments, not to *scale up*. CouchDB supports ACID semantics with eventual consistency, but only for single document access. Complex transactions that update multiple documents are not guaranteed to be atomic.

Moreover, CouchDB leverages Multi-version Concurrency Control (MVCC) to avoid locking on writes. This mechanism is known as copy-on-write that clones the entire traversal path from root to an appropriate leaf of the *B+tree* for each update. As discussed, our cloning scheme is expected to be faster and better in memory consumption.

A comparison between DStore, H-Store, HyPer and CouchDB is summarized in Table 1. Because DStore is in an early prototype state rather than a fully implemented system, we cannot perform any performance comparison between DStore and the presented systems. In the near future, we will finalize our comparison by performing more experiments on real-life workloads.

7 Summary

In this research report, we have introduced DStore, a document-oriented store. It is designed to scale up (vertically) in single server by adding more CPU resources and increasing the memory capacity. DStore targets the *Big Velocity* characteristic of Big Data that refers to the high speed of data accessing in storage system. DStore demonstrates fast and atomic transaction processing reflected in the update rate in data writing, while it also delivers high-throughput read accesses for analytical purposes. DStore adds support for atomicity for complex queries and does so with low overheads, property that has not been possible in document-oriented stores designed to scale-out.

In order to achieve its goals, DStore relies entirely on main memory for storing data. It leverages several key design principles, such as: single threaded-execution model, parallel index generation to leverage multi-core architectures, shadowing for concurrency control, and Stale Read support for high performance.

Our preliminary synthetic benchmarks demonstrate that DStore achieves high performance even under concurrency, where Read queries, Insert queries and Delete queries are performed in parallel. The experiments show low overheads for both reading and writing when increasing the number of concurrent readers. The measured processing rate was about 600,000 operations per second for each process. Moreover, DStore demonstrates good support for parallel index generations. Indeed, the processing rate does not drop down by a factor of 2 when doubling the number of indexes.

Contents

References

[1] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era (it's time for a complete rewrite)," in *VLDB '07: Proceedings of the 33rd international conference on very large data bases*, pp. 1150–1160, VLDB Endowment, 2007.

[2] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas, "Nobody ever got fired for using Hadoop on a cluster," in *HotCDP '12: Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, pp. 2:1–2:5, ACM, 2012.

[3] "CouchDB." `couchdb.apache.org`.

[4] "MongoDB." `www.mongodb.org`.

[5] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP through the looking glass, and what we found there," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on management of data*, pp. 981–992, ACM, 2008.

[6] D. G. Severance and G. M. Lohman, "Differential files: their application to the maintenance of large databases," *ACM Transactions on Database Systems (TODS)*, vol. 1, pp. 256–267, Sept. 1976.

[7] O. Rodeh, "B-trees, shadowing, and clones," *ACM Transactions on Storage*, vol. 3, no. 4, pp. 1–27, 2008.

[8] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, C. A. Soules, and A. Veitch, "Lazybase: trading freshness for performance in a scalable database," in *EuroSys '12: Proceedings of the 7th ACM european conference on Computer Systems*, pp. 169–182, ACM, 2012.

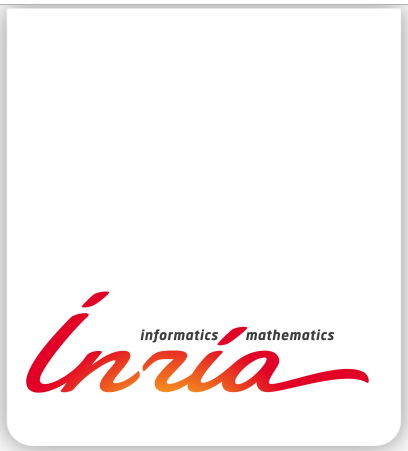
[9] S. Das, D. Agrawal, and A. E. Abbadi, "G-Store: a scalable data store for transactional multi key access in the Cloud," in *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pp. 163–174, ACM, 2010.

[10] "Boost." `www.boost.org`.

[11] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: a high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.

[12] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots," in *ICDE '11: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, pp. 195 –206, April 2011.

[13] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399